GO + SQLite

Go sqlite3

last modified August 24, 2023

Go sqlite3 tutorial shows how to work with sqlite3 database in Golang. The examples perform basic database operations.

```
$ go version
go version go1.18.1 linux/amd64
```

We use Go version 1.18.

SQLite is an embedded relational database engine. The documentation calls it a self-contained, serverless, zero-configuration and transactional SQL database engine. It is very popular with hundreds of millions copies worldwide in use today.

Go has the `sql` package which provides a generic interface around SQL (or SQL-like) databases. The sql package must be used in conjunction with a database driver.

Creating SQLite database

We can use the `sqlite3` command line tool to create a database and query for data.

```
$ sudo apt install sqlite3
```

We install the tool.

```
$ sqlite3 test.db
SQLite version 3.37.2 2022-01-06 13:25:41
Enter ".help" for usage hints.
sqlite>
```

We provide a parameter to the `sqlite3` tool; the `test.db` is the database name. It is a file on our disk. If it is present, it is opened. If not, it is created.

```
sqlite> .tables
sqlite> .exit
$ ls
test.db
```

The `.tables` command gives a list of tables in the `test.db` database. There are currently no tables. The `.exit` command terminates the interactive session of the `sqlite3` command line tool. The `ls` command shows the contents of the current working directory. We can see the `test.db` file. All data will be stored in this single file.

Go sqlite3 version

In the first example, we print the version of sqlite3.

`main.go`

```
package main


import (

    "database/sql"

    "fmt"

    "log"


    _ "github.com/mattn/go-sqlite3"

)
```

```
func main() {

    db, err := sql.Open("sqlite3", ":memory:")

    if err != nil {
        log.Fatal(err)
    }

    defer db.Close()

    var version string
    err = db.QueryRow("SELECT SQLITE_VERSION()").Scan(&version)

    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(version)
}
```

The program returns the version of sqlite3. The version is determined by executing the `SELECT SQLITE_VERSION()` statement.

```
_ "github.com/mattn/go-sqlite3"
```

When a package is imported prefixed with a blank identifier, the init function of the package is called. The function registers the driver.

```
db, err := sql.Open("sqlite3", ":memory:")
```

With `sql.Open`, we open a database specified by its database driver name and a driver-specific data source name. In our case, we connect to the in-memory database.

```
defer db.Close()
```

The `Close` function closes the database and prevents new queries from starting.

```
err = db.QueryRow("SELECT SQLITE_VERSION()").Scan(&version)
```

The `QueryRow` executes a query that is expected to return at most one row. The `Scan` function copies the column from the matched row into the `version` variable.

```
$ go run main.go
3.38.5
```

Go sqlite3 Exec

The `Exec` function executes a query without returning any rows.

`main.go`

```
package main


import (
```

```
    "database/sql"

    "fmt"

    "log"


    _ "github.com/mattn/go-sqlite3"

)


func main() {


    db, err := sql.Open("sqlite3", "test.db")


    if err != nil {

        log.Fatal(err)

    }


    defer db.Close()


    sts := `

DROP TABLE IF EXISTS cars;

CREATE TABLE cars(id INTEGER PRIMARY KEY, name TEXT, price INT);

INSERT INTO cars(name, price) VALUES('Audi',52642);

INSERT INTO cars(name, price) VALUES('Mercedes',57127);

INSERT INTO cars(name, price) VALUES('Skoda',9000);

INSERT INTO cars(name, price) VALUES('Volvo',29000);

INSERT INTO cars(name, price) VALUES('Bentley',350000);

INSERT INTO cars(name, price) VALUES('Citroen',21000);

INSERT INTO cars(name, price) VALUES('Hummer',41400);
```

```
INSERT INTO cars(name, price) VALUES('Volkswagen',21600);
`

    _, err = db.Exec(sts)


    if err != nil {

        log.Fatal(err)

    }



    fmt.Println("table cars created")

}
```

In the example, we create a new table.

```
db, err := sql.Open("sqlite3", "test.db")
```

We create a new file database.

```
    sts := `
DROP TABLE IF EXISTS cars;
CREATE TABLE cars(id INTEGER PRIMARY KEY, name TEXT, price INT);
INSERT INTO cars(name, price) VALUES('Audi',52642);
INSERT INTO cars(name, price) VALUES('Mercedes',57127);
INSERT INTO cars(name, price) VALUES('Skoda',9000);
INSERT INTO cars(name, price) VALUES('Volvo',29000);
INSERT INTO cars(name, price) VALUES('Bentley',350000);
INSERT INTO cars(name, price) VALUES('Citroen',21000);
INSERT INTO cars(name, price) VALUES('Hummer',41400);
INSERT INTO cars(name, price) VALUES('Volkswagen',21600);
`
```

These are SQL statements to create a new table.

```
_, err = db.Exec(sts)
```

The statements are executed with `Exec`.

Go sqlite3 select all rows with Query

The `Query` executes a query that returns rows, typically a SELECT. The optional arguments are for any placeholder parameters in the query.

main.go

```
package main


import (

    "database/sql"

    "fmt"

    "log"


    _ "github.com/mattn/go-sqlite3"

)


func main() {


    db, err := sql.Open("sqlite3", "test.db")


    if err != nil {

        log.Fatal(err)

    }
```

```
    defer db.Close()



rows, err := db.Query("SELECT * FROM cars")

    if err != nil {
        log.Fatal(err)
    }

    defer rows.Close()

    for rows.Next() {

        var id int
        var name string
        var price int

        err = rows.Scan(&id, &name, &price)

        if err != nil {
            log.Fatal(err)
        }

        fmt.Printf("%d %s %d\n", id, name, price)
    }
```

```
}
```

The example selects all rows from the `cars` table.

```
rows, err := db.Query("SELECT * FROM cars")
```

This is the SQL to select all rows.

```
for rows.Next() {
```

The `Next` prepares the next result row for reading with the `Scan` method. It returns true on success, or false if there is no next result row or an error happened while preparing it.

```
err = rows.Scan(&id, &name, &price)
```

We read the row into the variables.

```
fmt.Printf("%d %s %d\n", id, name, price)
```

The current row is printed.

```
$ go run main.go
1 Audi 52642
2 Mercedes 57127
3 Skoda 9000
4 Volvo 29000
5 Bentley 350000
6 Citroen 21000
7 Hummer 41400
8 Volkswagen 21600
```

Go sqlite3 prepared statement

With prepared statements, we use placeholders instead of directly writing the values into the statements. Prepared statements increase security and performance of database operations.

main.go

```
package main


import (

    "database/sql"

    "fmt"

    "log"


    _ "github.com/mattn/go-sqlite3"

)


func main() {


    db, err := sql.Open("sqlite3", "test.db")


    if err != nil {

        log.Fatal(err)

    }
```

```
    defer db.Close()

    stm, err := db.Prepare("SELECT * FROM cars WHERE id = ?")

    if err != nil {
        log.Fatal(err)
    }



  defer stm.Close()

    var id int
    var name string
    var price int

    cid := 3

    err = stm.QueryRow(cid).Scan(&id, &name, &price)

    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("%d %s %d\n", id, name, price)
}
```

We use a prepared statement to select a specific row.

```
stm, err := db.Prepare("SELECT * FROM cars WHERE id = ?")
```

The `Prepare` function creates a prepared statement for later queries or executions. The placeholder ? is later filled with value.

```
err = stm.QueryRow(cid).Scan(&id, &name, &price)
```

The value passed to the `QueryRow` function maps to the placeholder.
The `Scan` function copies the columns from the matched row into the variables.

```
$ go run main.go
3 Skoda 9000
```

It is also possible to do the prepared statement in one step go with `QueryRow`.

`main.go`

```
package main


import (

    "database/sql"

    "fmt"

    "log"


    _ "github.com/mattn/go-sqlite3"

)


func main() {
```

```
    db, err := sql.Open("sqlite3", "test.db")

    if err != nil {
        log.Fatal(err)
    }

    defer db.Close()

    cid := 3

    var id int
    var name string
    var price int

    row := db.QueryRow("SELECT * FROM cars WHERE id = ?", cid)
    err = row.Scan(&id, &name, &price)

    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("%d %s %d\n", id, name, price)
}
```

The example creates a prepared statement; this time with `QueryRow`.

Go sqlite3 affected rows

The `RowsAffected` returns the number of rows affected by an update, insert, or delete statements.

`main.go`

```
package main


import (
    "database/sql"
    "fmt"
    "log"

    _ "github.com/mattn/go-sqlite3"
)


func main() {

    db, err := sql.Open("sqlite3", "test.db")

    if err != nil {
        log.Fatal(err)
    }

    defer db.Close()

    res, err := db.Exec("DELETE FROM cars WHERE id IN (1, 2, 3)")

    if err != nil {
```

```
        log.Fatal(err)
    }

    n, err := res.RowsAffected()

    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("The statement has affected %d rows\n", n)
}
```

In the code example, we delete three rows with a DELETE SQL statement. Then we print the number of deleted rows with `RowsAffected`.

```
$ go run main.go
The statement has affected 3 rows
```

In this article we have worked with sqlite3 in Go.